TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

JUHA VEPSÄ

SONARQUBEN HYÖDYNTÄMINEN OHJELMISTOPROJEKTISSA

Kandidaatintyö

Tarkastaja: Yliopistonlehtori Terhi Kilamo
Jätetty tarkastettavaksi 16.5.2018

TIIVISTELMÄ

JUHA VEPSÄ: SonarQuben hyödyntäminen ohjelmistoprojektissa, Utilizing SonarQube in a software project
Tampereen teknillinen yliopisto
Kandidaatintyö, 21 sivua
Toukokuu 2018
Tietotekniikan koulutusohjelma
Pääaine: Ohjelmistotekniikka
Tarkastaja: Yliopiston lehtori Terhi Kilamo
Avainsanat: SonarQube, laatu, lähdekoodi

Ohjelmiston sisäinen rakenne ja laatu ovat ohjelmiston kehittäjän kannalta tärkeässä asemassa. Ohjelmiston lähdekoodin hyvä laatu edesauttaa ohjelmiston sujuvaa kehitystyötä. Tämä pitää samalla kehityskustannukset maltillisena pitkällä aikavälillä. Laadun arviointiin ja mittaamiseen on kehitetty työkaluja, joista tässä työssä esitellään yhtä. SonarQube on avoimeen lähdekoodiin perustuva työkalu, joka analysoi ohjelmiston lähdekoodia ja tuottaa tämän perusteella raportin lähdekoodin laadusta.

Ohjelmiston laadulle on kehitetty standardeja, jotka ottavat kantaa myös ohjelmiston sisäiseen laatuun. ISO 25010 -standardi sisältää määrittelyn ohjelmiston ylläpidettävyydelle, johon muun muassa lähdekoodin laatu vaikuttaa. Lähdekoodin laatua voidaan arvioida esimerkiksi automaattisia työkaluja käyttäen tai suorittamalla koodikatselmuksia.

Tässä työssä esitellään sekä ohjelmiston sisäiseen laatuun liittyvää teoriaa, että SonarQuben ominaisuuksia ja lisäksi analysoidaan SonarQuben avulla Vaadin-ohjelmisto. Työssä kuvataan analysointiprosessi ja sen tuottamat tärkeimmät tulokset Vaadinin osalta. Tulosten perusteella voidaan arvioida, että Vaadinin lähdekoodin ongelmien absoluuttinen määrä on kasvanut ohjelmistoprojektin edetessä. Johtopäätöksissä todetaan, että SonarQube soveltuu oikein säädettynä hyvin esimerkiksi projektijohdon apuvälineeksi projektin seurantaan.

SISÄLLYS

1. JOHDANTO

Nykyaikaisessa ohjelmistokehityksessä kehitettävän ohjelmiston lähdekoodin laatu on tärkeässä asemassa. Samaa lähdekoodia muokataan ja käsitellään useita kertoja kehityssyklien aikana. Heikko lähdekoodin laatu haittaa koodin parissa työskentelyä. Ohjelmistokehittäjiltä vie kauemmin aikaa ymmärtää monimutkaista, epäselvää ja hätäisesti kirjoitettua koodia, johon pitäisi tehdä muutoksia. Ylläpitämällä lähdekoodin laatua edesautetaan samalla ohjelmistokehittäjiä tekemään yleisesti laadukkaita ohjelmistotuotteita.

Lähdekoodin laadun seuraamiseksi on kehitetty paljon työkaluja. Työkalut analysoivat jollain tavalla kehitettävää ohjelmistoa ja antavat tämän perusteella palautetta ohjelmiston tilasta. Yksi tähän tarkoitukseen kehitetyistä työkaluista on SonarQube. SonarQube analysoi ohjelmiston lähdekoodia ja tuottaa raportin tämän laadusta. Tämä työ käsittelee SonarQuben käyttöä ohjelmistoprojektin yhteydessä ja sitä, miten lähdekoodin laatua voidaan arvioida SonarQuben avulla. Työssä analysoidaan avoimeen lähdekoodiin perustuva Vaadin-ohjelmisto, minkä perusteella SonarQuben käyttöä arvioidaan.

Tässä työssä esitellään ensin luvussa 2 määritelmiä ohjelmiston lähdekoodin laadulle. Luvussa 3 käydään läpi SonarQuben ominaisuuksia ja toimintaa. Luvussa 4 käsitellään analyysin ajamista SonarQubea käyttäen sekä sitä, miten analyysiä on mahdollista automatisoida. Luku 5 sisältää analyysin tulosten tarkastelun, ja luvussa 6 esitellään johtopäätökset SonarQuben käytöstä ohjelmistoprojektissa.

2. OHJELMISTON LÄHDEKOODIN LAATU

Ohjelmiston laatua voidaan tarkastella useista eri näkökulmista. Loppukäyttäjälle näkyvät usein ohjelmiston käytettävyyteen ja käyttökokemukseen vaikuttavat laatutekijät, kuten muun muassa käyttöliittymän selkeys, ulkonäkö tai ohjelmiston nopeus [1]. Ohjelmiston kehittäjän työn kannalta on sen sijaan tärkeämpää ohjelmiston lähdekoodin laatu, etenkin ohjelmiston aktiivisessa kehitysvaiheessa. Hyvä, selkeä lähdekoodin laatu parantaa ohjelmiston ylläpidettävyyttä, nopeuttaa ohjelmiston kehittämistä ja vähentää ohjelmointivirheiden määrää [2].

2.1 Ohjelmiston laadun määritelmät

ISO 25010 -standardi jakaa ohjelmiston laadun kahdeksaan kategoriaan, jotka jakaantuvat edelleen alakategorioihin [3]. Korkeimman tason kategoriat ovat seuraavat:

- funktionaalinen soveltuvuus
- suorituskyky
- yhteensopivuus
- käytettävyys
- luotettavuus
- turvallisuus
- siirrettävyys
- ylläpidettävyys.

Funktionaalinen soveltuvuus kuvaa sitä, kuinka hyvin ohjelmisto tai systeemi soveltuu sille suunniteltuihin käyttövaatimuksiin ja -tarpeisiin ohjelmiston suunnitellussa käyttöympäristössä. Suorituskyky kertoo ohjelmiston suorituskyvystä suhteessa käytössä oleviin resursseihin ohjelmiston suunnitellussa ajoympäristössä. Suorituskyky voi vaikuttaa esimerkiksi ohjelmiston laskentanopeuteen. Yhteensopivuus tarkoittaa ohjelmiston, systeemin tai komponentin kykyä kommunikoida muiden, samassa ajoympäristössä ajettavien ohjelmistojen, systeemien tai komponenttien kanssa.

Ohjelmiston, systeemin tai komponentin on kyettävä suorittamaan sille määritellyt tehtävät onnistuneesti kyseisessä ajoympäristössä. Käytettävyys tarkoittaa sitä, kuinka tyytyväisiä ohjelmiston loppukäyttäjät ovat määritellyissä käyttökonteksteissa ohjelmistoa käyttäessään ja kuinka tehokkaasti käyttäjät saavuttavat määritellyt tavoitteet näissä konteksteissa. Luotettavuus kuvaa ohjelmiston, systeemin tai komponentin kykyä suorittaa sille määritellyt toiminnot vakaasti ja onnistuneesti määritellyissä olosuhteissa määritellyn ajan verran. Luotettavuus viittaa esimerkiksi ohjelmiston kykyyn pysyä virheettömästi käynnissä. Turvallisuus kertoo, kuinka tehokkaasti ohjelmisto suojaa sen sisältämän informaation ja datan siten, että ohjelmistoa käyttävillä henkilöillä tai muilla ohjelmistokomponenteilla on kuitenkin riittävät käyttöoikeudet dataan tehtäviensä suorittamiseksi. Siirrettävyys tarkoittaa, kuinka tehokkaasti ohjelmisto voidaan siirtää yhdestä ajoympäristöstä toiseen.

Ylläpidettävyys liittyy vahvasti lähdekoodin laatuun, ja se kuvaa, kuinka tehokkaasti ohjelmistoa voidaan muokata esimerkiksi sen parantamiseksi ja korjaamiseksi tai ajoympäristön muututtua. Ylläpidettävyys jakaantuu viiteen alakategoriaan:

- modulaarisuus
- uudelleenkäytettävyys
- analysoitavuus
- muunneltavuus
- testattavuus.

Modulaarisuudella tarkoitetaan ohjelmiston rakenteen jakoa komponentteihin siten, että muutokset yhdessä komponentissa eivät vaikuta muiden komponenttien toimintaan. Uudelleenkäytettävyydellä tarkoitetaan sitä astetta, kuinka tehokkaasti jo olemassa olevia ohjelmakomponentteja voidaan hyödyntää eri systeemien ja uusien komponenttien luonnissa. Analysoitavuus kuvaa, kuinka helposti ohjelmistorakennetta ja lähdekoodia voidaan diagnosoida ja miten tehokkaasti lähdekoodista havaitaan muutoksia tarvitsevia osia. Muunneltavuudella kuvataan, kuinka helposti ohjelmistoa voidaan muokata luomatta vahingossa uusia haavoittuvuuksia tai vähentämättä ohjelman toiminnallisuutta. Testattavuudella tarkoitetaan, kuinka helposti ohjelmistolle voidaan määritellä testauskriteerit ja kuinka tehokkaasti kyseiset testit voidaan ohjelmistolle suorittaa.

CISQ:lla *(Consortium for IT Software Quality)* on oma määritelmänsä ohjelmiston laadulle [4]. Tämän määritelmän mukaisesti ohjelmiston laatu voidaan jakaa neljään kategoriaan:

- luotettavuus

- tehokkuus
- turvallisuus
- ylläpidettävyys.

Luotettavuus kuvaa ohjelmiston kykyä sietää virhetilanteita ja yleisesti ohjelmiston kykyä pysyä käynnissä. Tehokkuus tarkoittaa ohjelmiston nopeutta ja kykyä suorittaa esimerkiksi raskaita laskentaoperaatioita. Turvallisuudella tarkoitetaan ohjelmiston tietoturvaa ja alttiutta haavoittuvuuksille. Ylläpidettävyydellä kuvataan ohjelmiston lähdekoodin laatua ja esimerkiksi koodimuutosten tekemisen helppoutta.

2.2 Syitä ohjelmiston laadun seuraamiseen

Ohjelmiston ylläpitokustannukset voivat olla merkittävä osa ohjelmiston aiheuttamista kokonaiskustannuksista [5]. Nykyään ohjelmistoihin lisätään uusia ominaisuuksia julkaisun jälkeen, jolloin ohjelmiston ylläpidettävyys korostuu [6]. Heikko ylläpidettävyys hidastaa ohjelmistopäivitysten tekemistä ja lisää virheiden riskiä, mikä näkyy edelleen loppukäyttäjälle esimerkiksi ohjelmiston huonona toimivuutena. Ohjelmiston laadukas ja selkeä lähdekoodi helpottaa kehittäjien työtä ja parantaa ylläpidettävyyttä.

Ohjelmistoja päivitetään julkaisun jälkeen myös sieltä löytyvien virheiden korjaamiseksi. Ohjelmakoodista löytyvien virheiden korjaaminen on sitä kalliimpaa, mitä myöhemmässä vaiheessa kehitysprosessia ne havaitaan [7]. Tämän vuoksi lähdekoodin laadun aktiivinen seuraaminen projektin alusta alkaen ja mahdollisimman monen virheen korjaaminen ennen ohjelmiston julkaisua on kannattavaa.

Kehitystyön alla olevan ohjelmiston parissa työskentelee tyypillisesti useita henkilöitä samanaikaisesti. Tällöin myös lähdekoodin ylläpidon merkitys korostuu, sillä kaikkien lähdekoodin parissa työskentelevien on pystyttävä tehokkaasti tulkitsemaan ja kirjoittamaan kaikille yhteistä koodia. Yhteisesti sovitut koodauskäytännöt ja niitä valvovat työkalut edesauttavat ohjelmiston kehitystä. Samalla vähennetään myös työntekijän projektissa työskentelyn lopettamisesta aiheutuvaa haittaa, kun lopettaneen työntekijän kirjoittama lähdekoodi siirtyy muiden vastuulle.

Ohjelmistojen kehittäminen on usein tasapainoilua hyvin kirjoitetun, laadukkaan koodin ja määräaikojen välillä [8]. Kehitystyössä joudutaan usein tekemään ratkaisuja nopeuttaa ohjelmiston kehittämistä, esimerkiksi lähestyvän määräajan vuoksi, ohjelmiston sisäisen laadun kustannuksella. Tällöin pyritään kirjoittamaan ohjelmakoodia, joka täyttää funktionaaliset vaatimukset, mutta ei ole esimerkiksi yleisesti

hyväksyttyjen tyylisääntöjen mukaista. Ohjelmiston ylläpidettävyys kärsii tästä hetkellisesti, ja onkin tärkeää myöhemmin palata uudelleenkirjoittamaan nämä osuudet koodista. Oikein toteutettuna, tämänkaltainen *teknisen velan* hallinta on kuitenkin liiketoiminnan kannalta kannattavaa [9].

2.3 Ohjelmiston laadun mittaaminen

Ohjelmiston laadun määrittelemiseen on kehitetty standardeja [10][3] ja laadun mittaamiseen työkaluja, esimerkiksi SonarQube [11]. Työkalut voidaan jakaa joko staattisiin - tai dynamisiin työkaluihin [12]. Dynaamiset analysointityökalut analysoivat ohjelmistoa sen ajon aikana. Ohjelmaa ajetaan esimerkiksi tietyin lähtöarvoin ja muunnellen ohjelman suoritupolkua, kunnes saavutetaan haluttu ohjelman tila. Staattiset analysointityökalut analysoivat ohjelman lähde- tai binäärikoodia ja tuottavat tämän perusteella raportin ohjelmiston laadusta. Staattiset lähdekoodin analysointityökalut keskittyvät usein koodin laadun arvioimiseen tai koodissa sijaitsevien mahdollisten virheiden etsintään. Analysointityökalujen etuna on niiden nopeus ja ne voidaan usein konfiguroida toimimaan automaattisesti muiden ohjelmistokehitystyökalujen rinnalla, ilman että niitä tarvitsee erikseen suorittaa. Analysointityökalujen haittapuolena on niiden vaatimat tarkat konfiguraatiot ja mahdolliset virheelliset raportit.

Lähdekoodin laatua voidaan tarkastella ja mitata myös ihmisvoimin, esimerkiksi suorittamalla koodikatselmuksia [7]. Koodikatselmukset ovat työkalujen suorittamiin automaattisiin tarkastuksiin verrattuna usein työläämpiä suorittaa, mutta huolellisesti toteutettuna, ne tuottavat laadukasta tietoa lähdekoodista [13]. Koodikatselmus on ennaltasuunniteltu, kaavamainen prosessi. Koodikatselmuksessa ohjelmiston lähdekoodia luetaan läpi ihmisvoimin ja koodista pyritään löytämään esimerkiksi virheitä. Katselmuksen suorittavat siihen erityisesti nimetty ryhmä, johon voi kuulua esimerkiksi tarkasteltavan ohjelmiston parissa työskenteleviä henkilöitä. Katselmusryhmässä voi olla henkilöitä myös projektin ulkopuolelta. Katselmusprosessi on ennalta tarkasti määritelty, jossa ryhmä käy yhdessä lähdekoodia läpi segmentti kerrallaan. Katselmusprosessin aikana löytyvät virheet kirjataan ylös myöhemmin tapahtuvaa korjausta varten. Koodikatselmusten seurauksena, tarkasteltavan ohjelmiston lähdekoodista löydetään usein virheitä ja näiden korjaaminen parantaa lähdekoodin ja ohjelmiston laatua.

3. SONARQUBE

SonarQube on avoimeen lähdekoodiin perustuva, lähdekoodin analysointiin käytettävä työkalu [11]. SonarQubelle syötetään analysoitavan ohjelmiston lähdekoodi, jonka perusteella se tuottaa raportin lähdekoodin laadusta. SonarQube pystyy analysoimaan useita eri ohjelmointikieliä ja se on laajennettavissa sekä kaupallisilla, että avoimeen lähdekoodiin perustuvilla lisäosilla. SonarQube tukee useita eri käyttöjärjestelmiä, mutta se vaatii Javan asentamisen ajoympäristöönsä.

3.1 SonarQuben ominaisuudet

SonarQube on staattinen lähdekoodin analysointityökalu [11]. SonarQubelle voidaan antaa syötteenä esimerkiksi kehitteillä olevan ohjelmiston koko lähdekoodi. SonarQube tuottaa lähdekoodin perusteella raportin, joka sisältää tietoa muun muassa koodin sisältämästä teknisestä velasta, mahdollisista tietoturva-ongelmista sekä koodin yleisestä laadusta. Loppuraportti esittää paljon tietoa graafisesti erilaisten kuvien ja kuvaajien välityksellä. Analyysi voidaan ajaa tietyin väliajoin, jolloin SonarQube pystyy tuottamaan trendi-tyyppistä dataa lähdekoodin laadusta. SonarQube voidaan myös integroida osaksi jatkuvan julkaisun putkea, jolloin analyysi ajetaan automaattisesti esimerkiksi jokaisen uuden julkaistavan ohjelmistoversion kohdalla. SonarQubessa on mahdollista ylläpitää, seurata ja vertailla useita erillisiä projekteja samanaikaisesti.

SonarQube jaottelee koodista löytyvät virheet eri kategorioihin, joista merkittävimmät ovat *ohjelmointivirheet*, *haavoittuvuudet* sekä *koodihajut* (engl. code smell) [14]. Ohjelmointivirhe tarkoittaa koodissa olevaa mahdollista virhettä, joka saa ohjelman toimimaan esimerkiksi väärin tai epävakaasti. Haavoittuvuus tarkoittaa puolestaan ohjelmistossa olevaa mahdollista tietoturva-aukkoa, joka voi aiheuttaa esimerkiksi tietoturvauhan kyseiselle ohjelmistolle tai sitä ajavalle järjestelmälle. Koodihaju kuvaa ylläpito-ongelmaa lähdekoodissa, joka voi aiheuttaa ongelmia esimerkiksi lähdekoodin parissa työskenteleville. Koodihaju voi tarkoittaa esimerkiksi vaikeasti tulkittavissa olevaa koodisegmenttiä tai segmenttiä, jota muokattaessa, on tavallista suurempi riski tehdä virheitä koodiin. Koodihajuksi merkitty koodisegmentti voi

toimia funktionaalisesti täysin halutulla tavalla, mutta se on kirjoitettu esimerkiksi yleisesti hyväksyttyjen tyylioppaiden vastaisesti.

SonarQube tarjoaa myös *laatuportti* (engl. quality gate) -arvion [15], joka kertoo yhdellä arvolla, voiko analysoidun ohjelmiston julkaista tuotantoon. Laatuporttiarvio testaa analysoitua projektia ennaltamäärättyjä ehtoja vasten ja antaa näiden perusteella arvion ohjelmiston julkaisukelpoisuudesta. Käyttäjä pystyy itse määrittelemään laatuportin käyttämät ehdot. Ehdot voisivat olla esimerkiksi:

- Ei uusia vakavia haavoittuvuuksia havaittu.
- Testikattavuus uudessa koodissa yli 80%.
- Tekninen velka alle 50 päivää.

Laatuportin lisäksi SonarQube tarjoaa vastaavanlaiset, yhden arvon arviot, ohjelmiston luotettavuudelle, turvallisuudelle ja ylläpidettävyydelle [16]. Arviot annetaan asteikolla A-E, A:n ollessa paras ja E:n ollessa huonoin arvio. Luotettavuusarvio annetaan ohjelmistosta löytyneiden ohjelmointivirheiden perusteella. Turvallisuusarvio annetaan ohjelmistosta löytyneiden haavoittuvuuksien perusteella, ja vastaavasti ylläpidettävyysarvio annetaan ohjelmiston sisältämän teknisen velan perusteella. Näiden yhden arvon arvioiden perusteella SonarQuben ylläpitäjä voi halutessaan nopeasti tarkastaa projektin kunnon, syventymättä siihen sen tarkemmin.

3.2 SonarQuben toiminta

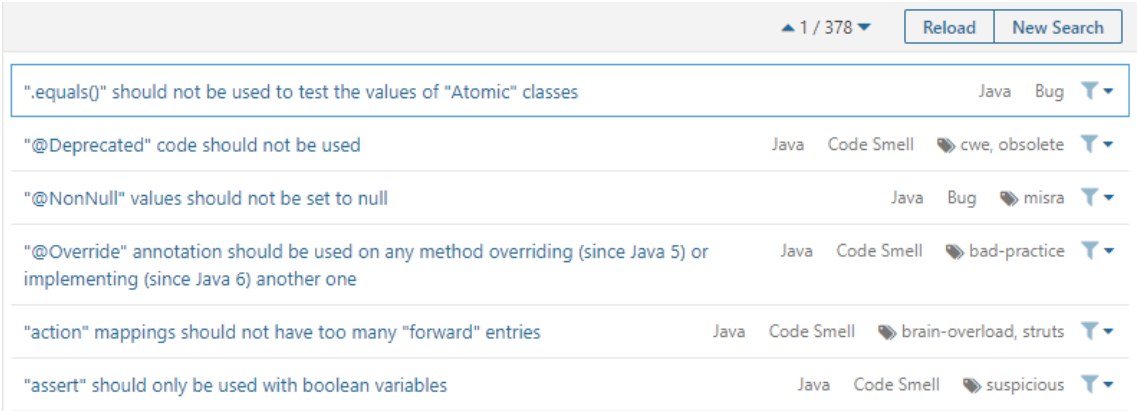
SonarQuben analyysi perustuu SonarQuben sisäisiin sääntöihin. Jokaista SonarQuben tukemaa ohjelmointikieltä kohden on SonarQubeen koodattu kyseistä ohjelmointikieltä koskevia sääntöjä. Jokainen sääntö sisältää tiedon muun muassa sääntörikkomuksen vakavuudesta ja mahdollisesti siitä aiheutuvan teknisen velan määrästä. SonarQuben oletusarvoiset säännöt perustuvat muun muassa yleisesti hyväksyttyihin hyviin ohjelmointitapoihin. Säännöt voivat liittyä esimerkiksi tunnettuihin tietoturvariskeihin, mahdollisesti ohjelmointivirheitä aiheuttavaan koodiin ja ohjelmointityyliin. Oletussääntöjä on mahdollista muokata oman tarpeen mukaan ja kokonaan uusia sääntöjä voidaan lisätä tai olemassa olevia sääntöjä poistaa. Kuvassa 3.1 on otos SonarQuben sisältämistä, Javaan liittyvistä oletussäännöistä.

Sääntöjä pääsee tarkastelemaan myös yksityiskohtaisemmin. Jokainen sääntö sisältää kuvauksen itse säännöstä, säännön tyypin, vakavuusasteen, tunnisteen, päivämäärän, jolloin sääntö on ensimmäisen kerran lisätty ajossa olevaan SonarQube-instanssiin, ohjelmointikielen, johon sääntö liittyy sekä säännön rikkomisesta aiheu-

tuvan teknisen velan määrän [17]. Sääntö sisältää myös perustelut kyseiselle säännölle sekä käytännön esimerkit sääntöä rikkovasta lähdekoodista ja sääntöä noudattavasta lähdekoodista. Säännön tyyppi on joko koodihaju, ohjelmointivirhe tai haavoittuvuus. Vakavuusaste on jokin viidestä [18]:

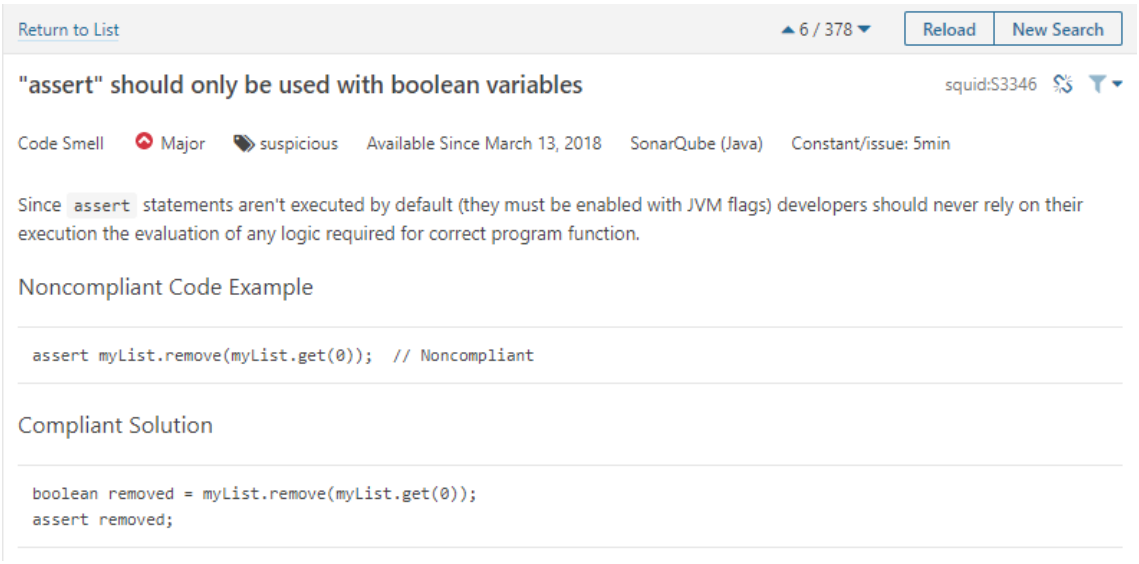
- estävä (engl. blocker)
- erittäin tärkeä (engl. critical)
- tärkeä (engl.major)
- vähäpätöinen (engl. minor)
- huomio (engl. info).

Estävä-tyypin vakavuusaste kuvaa lähdekoodissa olevaa ongelmaa, joka todennäköisesti vaikuttaa ohjelman toimintaan negatiivisesti. Estävä-tyypin ongelma voisi olla esimerkiksi muistivuoto. Tämän tyypin ongelmat eivät saisi päästä tuotantoon asti. Vakavuusasteeltaan erittäin tärkeillä ongelmilla on pieni todennäköisyys vaikuttaa ohjelman toimintaan haitallisesti. Tärkeät ja vähäpätöiset ongelmat liittyvät enemmän lähdekoodin laatuun ja voivat vaikuttaa kehittäjien tuottavuuteen. Huomio-tyypin vakavuusaste on vain jokin mahdollisesti huomiota tarvitseva seikka lähdekoodissa, eikä se tarkoita välttämättä ongelmaa tai virhettä. Kuvassa 3.2 on nähtävillä erään Javaan liittyvän säännön yksityiskohtaiset tiedot.



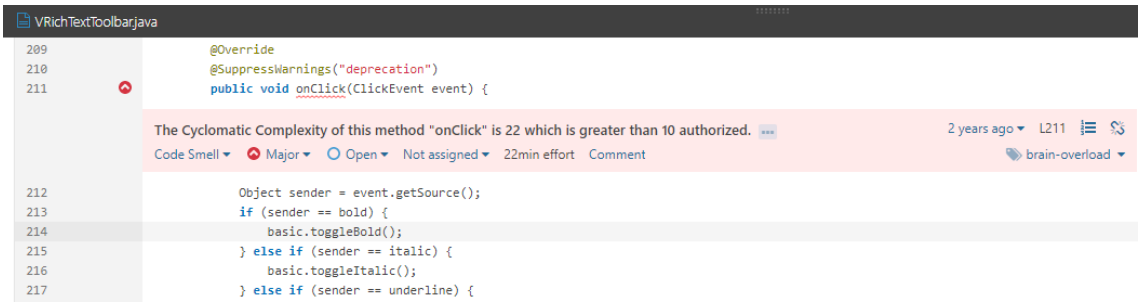
Kuva 3.1 *Otos SonarQuben sisältämistä, Javaan liittyvistä säännöistä.*

Analysoidessaan lähdekoodia, SonarQube käy läpi kaikki sille syötteenä annetut lähdekooditiedostot. Tiedostojen sisältämää lähdekoodia verrataan SonarQuben käytössä oleviin sääntöihin ja jokaista sääntörikkomusta kohden SonarQube tuottaa tiedon loppuraporttiin. Loppuraportissa pystyy tarkastelemaan sekä analysoidun lähdekoodin kokonaistilaa että yksittäisiä sääntörikkomuksia erikseen.



Kuva 3.2 *SonarQuben sisältämä sääntö, joka on tyypiltään koodihaju ja sen vakavuusaste on tärkeä.*

SonarQube jakaa lähdekoodista löytyneet ongelmat loppuraportissa eri kategorioihin, esimerkiksi luotettavuus, turvallisuus ja ylläpidettävyys. Ohjelmiston luotettavuusarvio kuvaa ohjelmistossa mahdollisesti olevien ohjelmointivirheiden määrää, turvallisuusarvio kuvaa mahdollisten haavoittuvuuksien määrää ja ylläpidettävyys kuvaa koodihajujen ja teknisen velan määrää. Lisäksi loppuraportissa on kategoriat muun muassa toisteisten koodirivien määrälle, analysoidun ohjelmiston koolle, koodin kompleksisuudelle ja dokumentoinnille. Kukin kategoria sisältää yksityiskohtaisempaa tietoa siihen liittyvistä asioista, esimerkiksi ohjelmiston kokoa voi tarkastella koodirivi-, funktio- tai luokka-määrien kautta. Kunkin kategorian alla olevia ongelmia ja tietoja pystyy jäljittämään lähdekoodissa eri tasoille. Esimerkiksi tietyn tiedoston tai ohjelmistokomponentin sisältämät tiedot saa haettua ja edelleen ongelmia voi jäljittää ongelman aiheuttavaan koodiriviin tai -segmenttiin saakka. Kuvassa 3.3 on nähtävillä erään koodihajun tarkka sijainti lähdekoodissa.



Kuva 3.3 *VRichTextToolbar.java -tiedosto sisältää koodihajuksi merkityn metodin rivillä 211.*

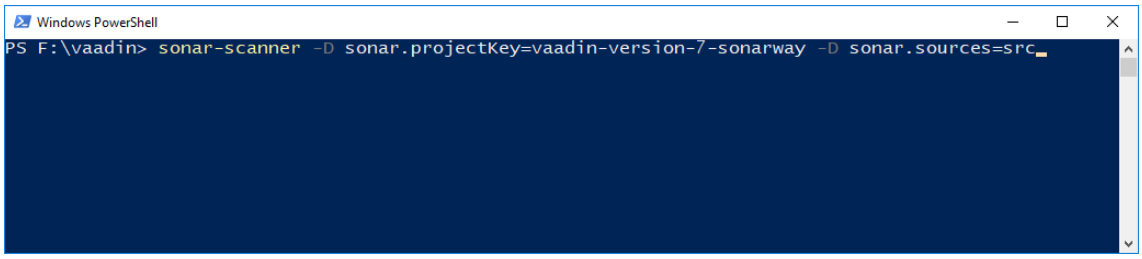
Eri kategorioiden kehitystä ohjelmistoprojektin edetessä pääsee myös tarkastelemaan, mikäli SonarQube-analyysejä on ajettu säännöllisesti. Loppuraportti näyttää muutokset esimerkiksi ohjelmointivirheiden tai teknisen velan määrässä, kun lähdekoodia on projektin edetessä muokattu.

4. OHJELMISTOPROJEKTIN ANALYSOINTI

Tässä työssä käytetään Vaadinin [19] avoimeen lähdekoodiin perustuvan Vaadin-työkalun lähdekoodia SonarQuben syötteenä. Vaadin on web-viitekehys [20], jonka avulla web-kehitystä voi tehdä Java-ohjelmointikielellä. SonarQube-analyysit ajetaan tässä työssä Windows-ympäristössä käyttäen *SonarQube Scanneria*. Analyysit ajetaan Vaadinin 7-versiolle, jolla on pidempi kehityshistoria kuin uudemmalla 8-versiolla. Jokaiselle Vaadinin 7-version *minor* ja *maintenance* -päivitykselle [21] ajetaan analyysi, jolloin SonarQubesta on nähtävillä trendi-tyyppistä dataa.

4.1 SonarQube-analyysin ajaminen

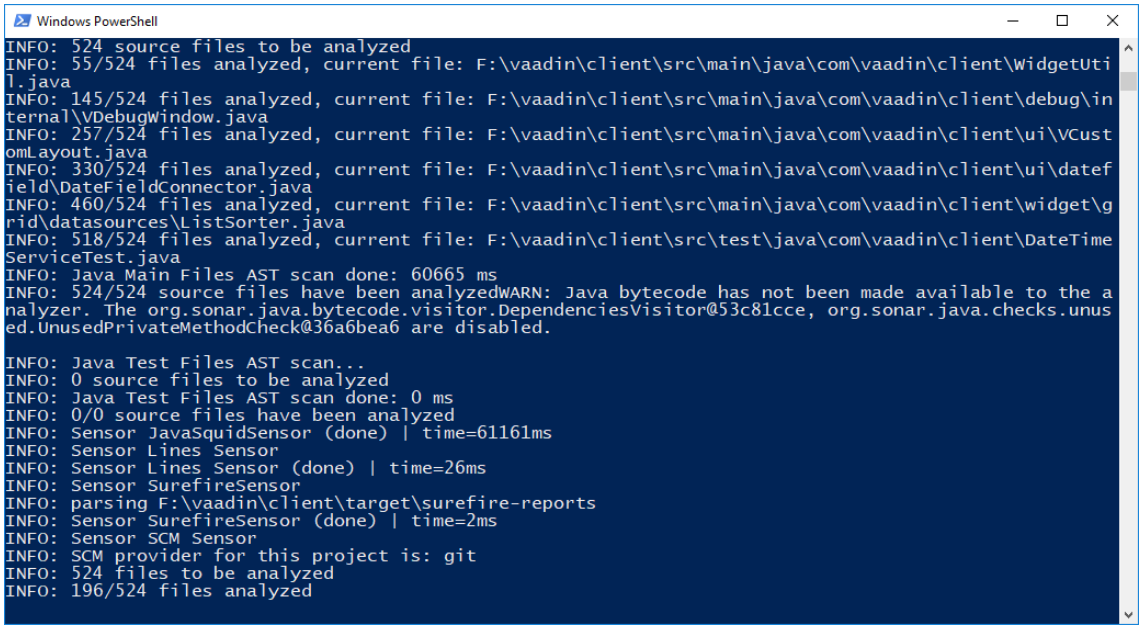
SonarQube Scanner on suoraviivaisin tapa analysoida ohjelmistoprojekti [22]. SonarQube Scanneria käytetään komentoriviltä ja se käynnistetään analysoitavan ohjelmiston juurihakemistossa. SonarQube Scannerille annetaan käynnistyksen yhteydessä projektiavain. Projektiavain yksilöi kunkin ylläpidettävän analysointiprojektin SonarQuben tietokannassa ja mahdollistaa näin usean eri projektin ylläpitämisen samanaikaisesti. Scannerille voidaan käynnistyksen yhteydessä syöttää myös muita analyysia ohjaavia parametrejä. Analysointiparametrejä voidaan antaa Scannerille myös erillisestä konfiguraatiotiedostosta. Kuvassa 4.1 Scanneri käynnistetään ja sille määritellään projektiavain sekä kohdehakemisto, jolle analyysi ajetaan.



Kuva 4.1 *SonarQube Scannerin ajaminen Windows-ympäristössä.*

SonarQuben analysointinopeus riippuu paljon analysoitavan kohteen koosta sekä analysoinnin suorittavan koneen tai palvelun resursseista. Tässä työssä Vaadinille

tehtyjen 84 analyysin läpi ajaminen vei useita tunteja. Tavallisemmassa käyttötapauksessa kuitenkaan harvoin analysoidaan koko versiohistoriaa kerralla, vaan SonarQube on liitetty osaksi julkaisuputkea ja dataa kertyy sitä mukaa, kun ohjelmistoa kehitetään. Kuvassa 4.2 on nähtävillä SonarQube Scannerin tuottamia tulosteita analysoinnin kulusta.



Kuva 4.2 *SonarQube Scanner analysoi lähdekoodia.*

SonarQube luo analyysien perusteella tietokannan, joka sisältää analyysin tuottamat tulokset. Analyysin tuloksia voidaan tarkastella web-selaimella, ottamalla yhteys paikallisesti käynnissä olevaan SonarQube-palvelimeen. Tulokset on esitetty visuaalisesti helposti havainnoitavassa muodossa, käyttäen paljon kuvia ja taulukoita. Pitkältä ajalta kerätystä datasta on nähtävillä koko projektin historia ja kehityskulku.

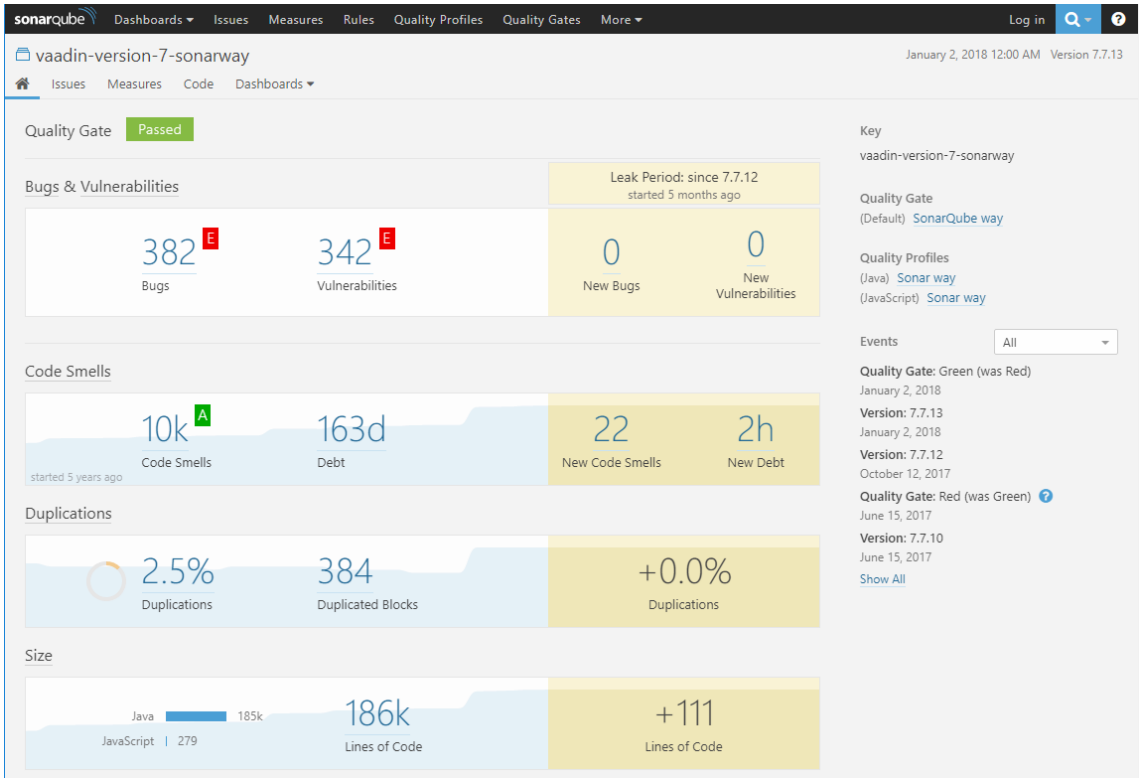
4.2 SonarQube Scannerin automatisointi

SonarQube Scannerin käyttäminen on yksinkertaisin ja nopein tapa analysoida lähdekoodi [22]. SonarQube Scannerin käyttämisestä komentoriviltä tulee kuitenkin työlästä, jos suoritettavia analyyseja on esimerkiksi kymmeniä. Tässä työssä analysoitiin 84 peräkkäistä Vaadin 7-version päivitystä eli myös analyysi ajettiin yhteensä 84 kertaa. Tämä olisi tarkoittanut analysoitavan version hakemista versionhallinnasta ja SonarQube Scannerin käynnistämistä eri parametreilla 84 kertaa manuaalisesti. Tämän automatisoimiseksi kirjoitettiin yksinkertainen Python-skripti, joka haki

versionhallinnasta analysointivuorossa olevan Vaadinin version, syötti SonarQube Scannerille oikeat analysointiparametrit ja käynnisti analyysin. Skripti odotti, että analyysi on suoritettu onnistuneesti ja tämän jälkeen skripti toisti itsensä, muuttaen analysoitavaa Vaadinin versiota ja analysointiparametreja. Skripti ajoi itseään niin kauan, kunnes kaikki haluttu materiaali oli analysoitu. Tämän ansiosta skriptin pystyi jättämään ajoon esimerkiksi yön yli ja tarvittaessa analyysien kokonaan uudelleen ajaminen ei tuottanut paljoa ylimääräistä työtä.

5. TULOKSET

SonarQube-analyysit ajettiin Vaadinin 7-versiolle jokaisen julkisen ohjelmistopäivityksen yhteydessä, jolloin saatiin trendi-tyyppistä dataa ohjelmiston kehitysvaiheista. Ensimmäinen analyysi on ajettu versiolle 7.0.4, joka on julkaistu huhtikuussa 2013 ja viimeisimpänä on analysoitu versio 7.7.13, joka on julkaistu tammikuussa 2018. Analyysit ajettiin SonarQube Scannerilla ja analyyseissä käytettiin SonarQubelle laadittuja Javan oletussääntöjä. Useiden kymmenien analyysien ajamisessa hyödynnettiin itse kirjoitettua Python-skriptiä, joka lukee tekstitiedostosta vaadittavat analysointiparametrit, hakee versionhallinnasta vuorossa olevan Vaadinin version analysoitavaksi ja ajaa tälle SonarQube Scannerin automaattisesti.



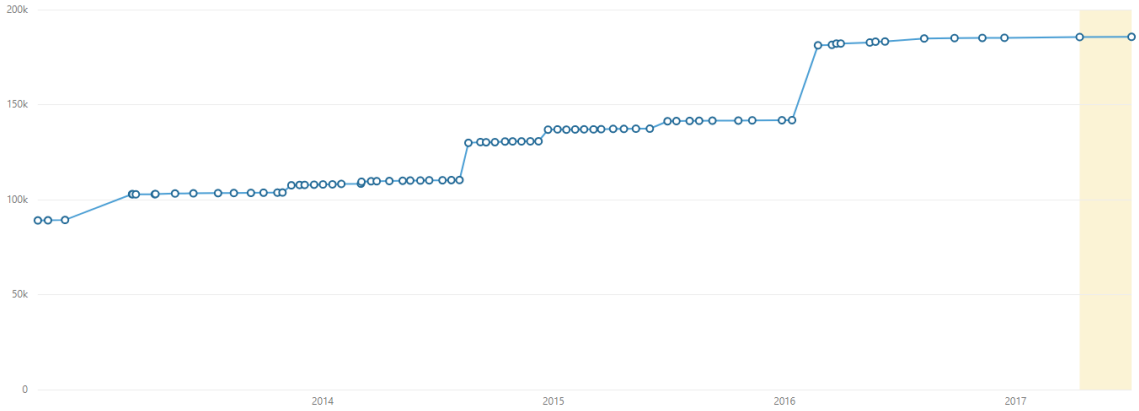
Kuva 5.1 *Yleisnäkymä projektin tilasta. Projektin tunnuslukuja valkoisella pohjalla ja muutokset edelliseen versioon kellertävällä pohjalla.*

Kuvassa 5.1 on esitetty yleisnäkymä projektin tilasta analyysien jälkeen. Yleisnäky-

mä näyttää kokonaiskuvan muun muassa ohjelmistossa olevista mahdollisista ohjelmointivirheistä, haavoittuvuuksista, koodihajuista sekä teknisestä velasta. Näkymä näyttää myös yhden arvon arviot laatuportille sekä ohjelmiston luotettavuudelle, turvallisuudelle ja ylläpidettävyydelle. Yleisnäkymä näyttää myös projektin tilassa tapahtuneet muutokset tuoreimman analyysin ja halutun vertailukohdan välillä. Tässä tapauksessa vertailukohta on edellinen versio ohjelmistosta, eli versio 7.7.12.

Yleisnäkymästä voidaan havaita, että analysoitavassa ohjelmistoprojektissa on tällä hetkellä noin 186 tuhatta riviä ohjelmakoodia. Ohjelmisto sisältää SonarQuben analyysin mukaan 382 ohjelmointivirhettä ja 342 haavoittuvuutta. Erilaisia koodihajuja SonarQube on erottanut yhteensä noin 10 tuhatta kappaletta ja teknistä velkaa ohjelmistossa on 163 päivän verran. Toisteista koodia on ohjelmistossa noin 2,5%. Laatuportti läpäistään ja ohjelmiston ylläpidettävyysarvio on paras mahdollinen A, mutta sekä ohjelmiston luotettavuus että turvallisuus saavat huonoimman mahdollisen arvion E.

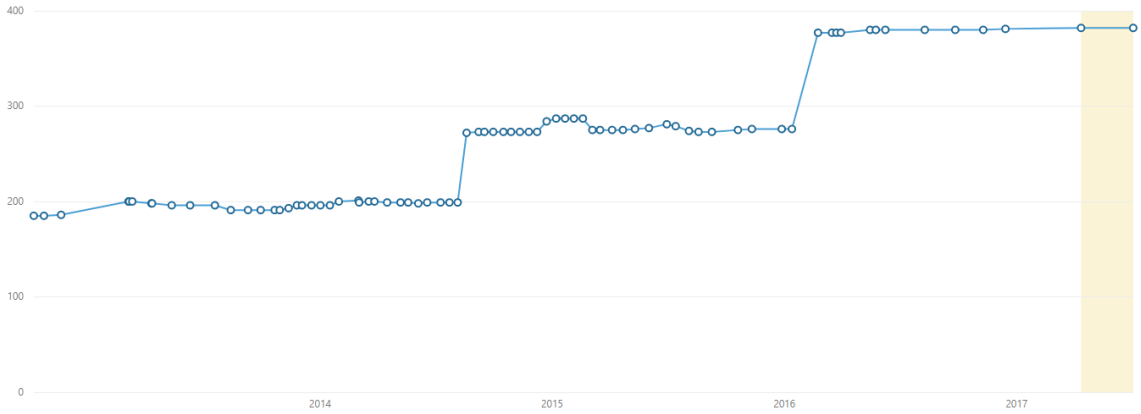
Tarkastellessa projektin ohjelmakoodin määrää läpi projektin, voidaan havaita sen kasvaneen huomattavasti sitten tarkastelujakson alun. Kuvassa 5.2 on nähtävillä koodirivien määrän kehitys läpi projektin. Ensimmäisessä analysoidussa versiossa ohjelmakoodia on yhteensä noin 89 000 riviä. Viimeisimmässä versiossa koodirivejä on noin 186 000 kpl. Kehityksessä on havaittavissa paikoin suurempia muutoksia koodin määrässä. Nämä ovat tyypillisesti Vaadinin nk. minor-päivityksiä eli siirtyminen esimerkiksi versiosta 7.6 versioon 7.7.



Kuva 5.2 *Projektin koodirivien määrän kehitys.*

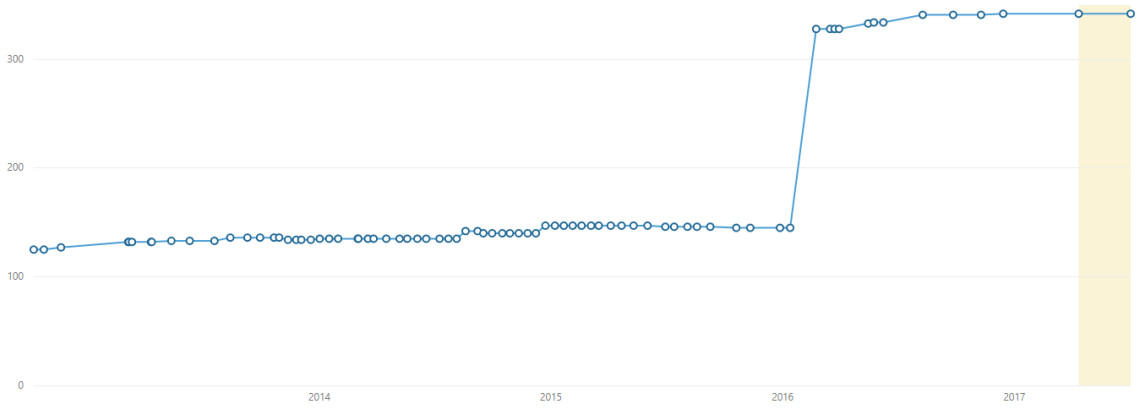
Tarkastellessa ohjelmistosta löytyvien ohjelmointivirheiden määrää, voidaan havaita että ohjelmointivirheiden määrä on kasvanut läpi projektin. Kuvassa 5.3 on nähtävillä ohjelmointivirheiden määrän kehitys läpi projektin. Ohjelmointivirheitä oli tarkasteluvaiheen alussa 185 kpl ja lopussa niitä oli yhteensä 382 kpl. Verrattaes-

sa ohjelmointivirheiden määrän kehitystä koodirivien määrän kehityksen kanssa, voidaan havaita, että ohjelmointivirheet kasvavat yhdessä koodirivien määrän kasvaessa. Suurimmat muutokset ovat jälleen isoissa ohjelmistopäivityksissä, joissa on tullut myös paljon uutta ohjelmakoodia. Kuvasta 5.3 voidaan havaita, että ohjelmointivirheitä on projektin edetessä paikoin myös korjattu ja niiden määrä on näin vähentynyt, mutta trendi on ollut koko ajan silti kasvava.



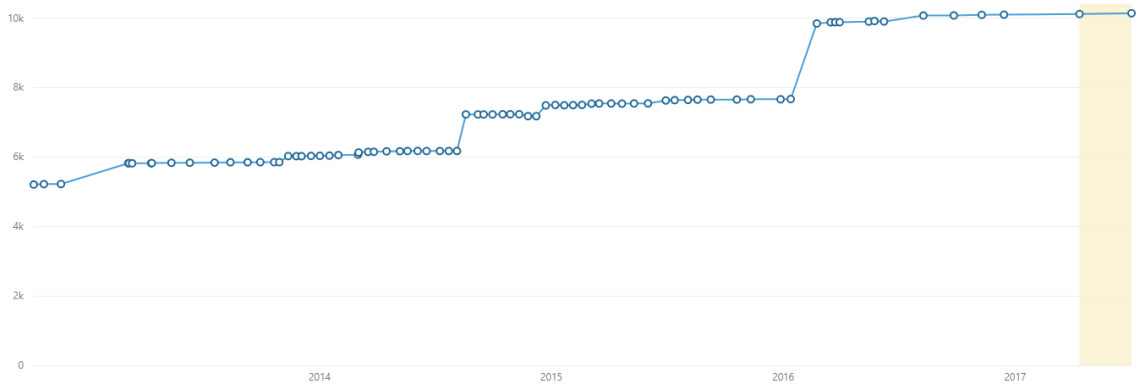
Kuva 5.3 *Ohjelmointivirheiden määrän kehitys.*

Projektista löytyneiden haavoittuvuuksien määrä on myös noussut tarkastelujakson alun jälkeen. Kuvassa 5.4 on nähtävillä projektin haavoittuvuuksien määrän kehitys. Tarkastelujakson alussa haavoittuvuuksia oli 125 kpl ja lopussa niitä oli 342. Haavottuvuuksien määrän kehitys eroaa hieman ohjelmointivirheiden kehityksestä, joka mukailee koodirivien määrän kehitystä. Haavoittuvuuksien määrä pysyy pitkään lähes ennallaan versioon 7.6.8 saakka. Tämän jälkeen on julkaistu versio 7.7.0, jonka yhteydessä on tullut runsaasti uutta ohjelmakoodia ja tämän myötä myös uusia haavoittuvuuksia.



Kuva 5.4 *Haavoittuvuuksien määrän kehitys.*

Koodihajujen määrä on myös noussut projektin tarkastelujakson aikana. Tarkastelujakson alussa koodihajuja oli noin 5200 kpl ja lopussa niitä oli noin 10 100 kpl. Kuva 5.5 havainnollistaa koodihajujen määrän kehitystä, ja se on hyvin samankaltainen koodirivien määrän kehityksen kanssa. Suurimmat muutokset koodihajujen määrässä tapahtuvat isojen päivitysten yhteydessä, jolloin koodihajuja tulee lisää. Koodihajujen määrä ei juurikaan vähene missään vaiheessa projektin aikana.



Kuva 5.5 *Koodihajujen määrän kehitys.*

Kaikissa edellä esitetyissä kuvaajissa on nähtävillä merkittävä muutos Vaadinin päivityttyä versiosta 7.6.8 versioon 7.7.0 vuoden 2016 aikana. Päivityksen yhteydessä on tullut uutta ohjelmakoodia noin 40 tuhatta riviä lisää. Päivitys näkyy samalla myös ongelmien lisääntymisenä. Toinen merkittävä muutos, mikä näkyy etenkin koodirivien, ohjelmointivirheiden sekä koodihajujen määrissä on vuoden 2015 aikana tehty päivitys versiosta 7.3.10 versioon 7.4.0.

6. JOHTOPÄÄTÖKSET

Vaadin-ohjelmisto on kasvanut lähes 100 000 koodirivin verran vajaan viiden vuoden tarkastelujakson aikana. Samalla myös SonarQuben löytämät ongelmat lähdekoodissa ovat kasvaneet. Kaikki SonarQuben mittaamat merkittävimmät ongelmat; ohjelmointivirheet, haavoittuvuudet ja koodihajut, ovat kasvaneet ohjelman koon kasvaessa. Tämä on toisaalta odotettua: ihmisten kehittäessä suuria määriä uutta ohjelmakoodia, joukkoon eksyy vääjäämättä myös virheitä. Suuria korjausliikkeitä ongelmien vähentämiseksi, esimerkiksi Vaadinin ylläpitopäivitysten yhteydessä, ei ole kuitenkaan SonarQuben tulosten perusteella tehty.

Merkittävä tekijä SonarQuben tulosten tulkinnassa on myös sen analyysien yhteydessä käyttämä sääntöjoukko. Tässä työssä sääntöjoukkona oli SonarQubelle laaditut Javan oletussäännöt. Oletussäännöt voivat poiketa projektissa käytettävistä säännöistä ja koodaustyyleistä, jolloin SonarQuben antamat tulokset ovat mahdollisesti epätarkkoja. SonarQuben oletussäännöissä voi olla esimerkiksi sääntö, joka ei ole relevantti Vaadin-ohjelmistoprojektille. Tällöin SonarQuben tuloksissa voi näkyä, että esimerkiksi ohjelmointivirheitä ei korjattaisi vaikka todellisuudessa kyse on virheellisestä raportista. Tarkempien tulosten takaamiseksi, SonarQuben käyttämää sääntöjoukkoa on syytä muokata analysoitavalle projektille sopivaksi.

SonarQuben tuottamien tulosten tarkastelu on tehty helpoksi. Tarkasteltavan projektin kokonaiskuva on helppo hahmottaa avainlukujen ja kuvaajien avulla. SonarQuben keräämä historiatieto näyttää projektin kehittymisen eri arvojen osalta. SonarQuben avulla projektista on mahdollista tunnistaa esimerkiksi komponentteja, jotka sisältävät paljon ongelmia. Kokonaiskuvan hahmottamisen lisäksi, projektia pystyy tarkastelemaan hyvin ykstyiskohtaisesti aina lähdekooditasolle saakka. SonarQube soveltuu esimerkiksi projektijohdon käytettäväksi, jolloin he pystyvät seuraamaan projektin kuntoa konkreettisesti.

SonarQuben käyttöönotto ja yksinkertaisten analyysien ajaminen on helppoa ja nopeaa. SonarQube Scannerin käyttäminen on vaivatonta. Suurilla analysointimäärillä SonarQube Scannerin käyttäminen käy kuitenkin nopeasti hyvin työlääksi. Scanneria pystyy kuitenkin jonkin verran automatisoimaan ja esimerkiksi Scannerin vaa-

timia komentoja voi suorittaa esimerkiksi skriptin avulla. SonarQube tukee myös integrointia jatkuvan julkaisun putkeen, mikä on teollisuudessa käytännöllisin tapa käyttää SonarQubea.

LÄHTEET

[1] N. Bevan, "Quality in use: Meeting user needs for quality," *Journal of Systems and Software*, vol. 49, no. 1, pp. 89–96, 1999.

[2] D. L. Parnas, "Software aging," in *Proceedings of 16th International Conference on Software Engineering*, May 1994, pp. 279–287.

[3] *ISO/IEC 25010, System and software quality models*, International Organization for Standardization, 2011.

[4] *CISQ Code Quality Standards*, Consortium for IT Software Quality, 2016. Saatavissa: http://it-cisq.org/standards/code-quality-standards/. (Viitattu: 17.10.2017)

[5] D. Coleman, D. Ash, B. Lowther, and P. Oman, "Using metrics to evaluate software system maintainability," *Computer*, vol. 27, no. 8, pp. 44–49, 1994.

[6] K. K. Aggarwal, Y. Singh, and J. K. Chhabra, "An integrated measure of software maintainability," in *Annual Reliability and Maintainability Symposium. 2002 Proceedings (Cat. No.02CH37318)*, 2002, pp. 235–241.

[7] M. E. Fagan, "Design and code inspections to reduce errors in program development," *IBM Systems Journal*, vol. 38, no. 2, pp. 258–287, 1999.

[8] M. Fowler. Technical debt. Saatavissa: https://martinfowler.com/bliki/TechnicalDebt.html. (Viitattu: 25.10.2017)

[9] E. Allman, "Managing technical debt," *Queue*, vol. 10, no. 3, pp. 10–17, mar 2012.

[10] Iso 9126 software quality characteristics. Saatavissa: http://www.sqa.net/iso9126.html. (Viitattu: 27.10.2017)

[11] Sonarqube. Saatavissa: https://www.sonarqube.org/. (Viitattu: 29.6.2017)

[12] P. Godefroid, P. de Halleux, A. V. Nori, S. K. Rajamani, W. Schulte, N. Tillmann, and M. Y. Levin, "Automating software testing using program analysis," *IEEE Software*, vol. 25, no. 5, pp. 30–37, 2008.

[13] G. W. Russell, "Experience with inspection in ultralarge-scale development," *IEEE Software*, vol. 8, no. 1, pp. 25–31, Jan 1991.

[14] Sonarqube documentation - concepts. Saatavissa: https://docs.sonarqube.org/display/SONAR/Concepts. (Viitattu: 9.4.2018)

[15] Sonarqube documentation - quality gates. Saatavissa: https://docs.sonarqube.org/display/SONAR/Quality+Gates. (Viitattu: 26.4.2018)

[16] Sonarqube documentation - metric definitions. Saatavissa: https://docs.sonarqube.org/display/SONAR/Metric+Definitions. (Viitattu: 26.4.2018)

[17] Sonarqube documentation - rules. Saatavissa: https://docs.sonarqube.org/display/SONAR/Rules. (Viitattu: 19.4.2018)

[18] Sonarqube documentation - issues. Saatavissa: https://docs.sonarqube.org/display/SONAR/Issues. (Viitattu: 19.4.2018)

[19] Vaadin. Saatavissa: https://vaadin.com/. (Viitattu: 17.4.2018)

[20] Vaadin framework. Saatavissa: https://vaadin.com/framework. (Viitattu: 17.4.2018)

[21] Vaadin roadmap. Saatavissa: https://vaadin.com/roadmap. (Viitattu: 14.3.2018)

[22] Sonarqube documentation - analyzing with sonarqube scanner. Saatavissa: https://docs.sonarqube.org/display/SCAN/Analyzing+with+SonarQube+Scanner. (Viitattu: 14.3.2018)